A PROCESSOR WITH A SPLIT STACK

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims priority to U.S. Provisional Application Serial No. 60/400,391 titled "JSM Protection," filed July 31, 2002, incorporated herein by reference. This application also claims priority to EPO Application No. 03291908.6, filed July 30, 2003 and entitled "A Processor With A Split Stack," incorporated herein by reference. This application also may contain subject matter that may relate to the following commonly assigned co-pending applications incorporated herein by reference: "System And Method To Automatically Stack And Unstack Java Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35422 (1962-05401); "Memory Management Of Local Variables," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35423 (1962-05402); "Memory Management Of Local Variables Upon A Change Of Context," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35424 (1962-05403); "Using IMPDEP2 For System Commands Related To Java Accelerator Hardware," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35426 (1962-05405); "Test With Immediate And Skip Processor Instruction," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35427 (1962-05406); "Test And Skip Processor Instruction Having At Least One Register Operand," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35248 (1962-05407); "Synchronizing Stack Storage," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35429 (1962-05408); "Methods And Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35430

(1962-05409); "Write Back Policy For Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35431 (1962-05410); "Methods And Apparatuses For Managing Memory," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35432 (1962-05411); "Mixed Stack-Based RISC Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35433 (1962-05412); "Processor That Accommodates Multiple Instruction Sets And Multiple Decode Modes," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35434 (1962-05413); "System To Dispatch Several Instructions On Available Hardware Resources," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35444 (1962-05414); "Micro-Sequence Execution In A Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35445 (1962-05415); "Program Counter Adjustment Based On The Detection Of An Instruction Prefix," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35452 (1962-05416); "Reformat Logic To Translate Between A Virtual Address And A Compressed Physical Address," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35460 (1962-05417); "Synchronization Of Processor States," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35461 (1962-05418); "Conditional Garbage Based On Monitoring To Improve Real Time Performance," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35485 (1962-05419); "Inter-Processor Control," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35486 (1962-05420); "Cache Coherency In A Multi-Processor System," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35637 (1962-05421); "Concurrent Task Execution In A Multi-Processor, Single Operating System Environment," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35638 (1962-05422); and "A Multi-Processor Computing System Having A Java Stack Machine And A RISC-

Based Processor," Serial No. _____, filed July 31, 2003, Attorney Docket No. TI-35710 (1962-05423).

BACKGROUND OF THE INVENTION

Technical Field of the Invention

[0002] The present invention relates generally to processors and more particularly to a processor capable of executing a stack-based instruction set and a non-stack based instruction set.

Background Information

[0003] Many types of electronic devices are battery operated and thus preferably consume as little power as possible. An example is a cellular telephone. Further, it may be desirable to implement various types of multimedia functionality in an electronic device such as a cell phone. Examples of multimedia functionality may include, without limitation, games, audio decoders, digital cameras, etc. It is thus desirable to implement such functionality in an electronic device in a way that, all else being equal, is fast, consumes as little power as possible and requires as little memory as possible. Improvements in this area are desirable.

BRIEF SUMMARY

[0004] Methods and apparatuses are disclosed for implementing a multi-stack processor. In some embodiments, the processor includes a main stack and a micro-stack. The micro-stack preferably is implemented in the core of the processor, whereas the main stack may be implemented in areas that are external to the core of the processor. Operands are preferably provided to an arithmetic logic unit (ALU) by the micro-stack, and in some cases, operands may be fetched from the main stack. By optimizing the size of the micro-stack, the number of operands fetched from the main stack may be reduced, and consequently the processor's power consumption may be reduced.

NOTATION AND NOMENCLATURE

[0005] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, semiconductor companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to...." Also, the term "couple" or "couples" is intended to mean either an indirect or direct connection. Thus, if a first device couples to a second device, that connection may be through a direct connection, or through an indirect connection via other devices and connections.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] For a more detailed description of the preferred embodiments of the present invention, reference will now be made to the accompanying drawings, wherein:

[0007] Figure 1 shows a diagram of a system in accordance with preferred embodiments of the invention and including a Java Stack Machine ("JSM") and a Main Processor Unit ("MPU");

[0008] Figure 2 shows a block diagram of the JSM of Figure 1 in accordance with preferred embodiments of the invention;

[0009] Figure 3 shows various registers used in the JSM of Figures 1 and 2;

[0010] Figures 4A-C depict stack management in the event of an overflow condition;

[0011] Figure 5A-C depict stack management in the event of an underflow condition; and

[0012] Figure 6 depicts an exemplary embodiment of the system described herein.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0013] The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims, unless otherwise specified. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

[0014] The subject matter disclosed herein is directed to a programmable electronic device such as a processor. The processor described herein is particularly suited for executing Java™ bytecodes or comparable, code. As is well known, Java is particularly suited for embedded applications. Java is a relatively "dense" language meaning that on average each instruction may perform a large number of functions compared to various other programming languages. The dense nature of Java is of particular benefit for portable, battery-operated devices that preferably include as little memory as possible to save space and power. The reason, however, for executing Java code is not material to this disclosure or the claims that follow. The processor described herein may be used in a wide variety of electronic systems. By way of example and without limitation, the Java-executing processor described herein may be used in a portable, battery-operated cell phone. Further, the processor advantageously includes one or more features that reduce the amount of power consumed by the Java-executing processor.

[0015] Referring now to Figure 1, a system 100 is shown in accordance with a preferred embodiment of the invention. As shown, the system includes at least two processors 102 and 104. Processor 102 is referred to for purposes of this disclosure as a Java Stack Machine ("JSM") and

processor 104 may be referred to as a Main Processor Unit ("MPU"). System 100 may also include an external memory 106 coupled to both the JSM 102 and MPU 104 and thus accessible by both processors. The external memory 106 may exist on a separate chip than the JSM 102 and the MPU 104. At least a portion of the external memory 106 may be shared by both processors meaning that both processors may access the same shared memory locations. Further, if desired, a portion of the external memory 106 may be designated as private to one processor or the other. System 100 also includes a Java Virtual Machine ("JVM") 108, compiler 110, and a display 114. The JSM 102 preferably includes an interface to one or more input/output ("I/O") devices such as a keypad to permit a user to control various aspects of the system 100. In addition, data streams may be received from the I/O space into the JSM 102 to be processed by the JSM 102. Other components (not specifically shown) may include, without limitation, a battery and an analog transceiver to permit wireless communications with other devices. As noted above, while system 100 may be representative of, or adapted to, a wide variety of electronic systems, an exemplary electronic system may comprise a battery-operated, mobile cell phone.

[0016] As is generally well known, Java code comprises a plurality of "bytecodes" 112. Bytecodes 112 may be provided to the JVM 108, compiled by compiler 110 and provided to the JSM 102 and/or MPU 104 for execution therein. In accordance with a preferred embodiment of the invention, the JSM 102 may execute at least some, and generally most, of the Java bytecodes. When appropriate, however, the JSM 102 may request the MPU 104 to execute one or more Java bytecodes not executed or executable by the JSM 102. In addition to executing Java bytecodes, the MPU 104 also may execute non-Java instructions. The MPU 104 also hosts an operating system ("O/S") (not specifically shown), which performs various functions including system memory management, the system task management that schedules the JVM 108 and most or all other native

tasks running on the system, management of the display 114, receiving input from input devices, etc. Without limitation, Java code may be used to perform any one of a variety of applications including multimedia, games or web based applications in the system 100, while non-Java code, which may comprise the O/S and other native applications, may still run on the system on the MPU 104.

[0017] The JVM 108 generally comprises a combination of software and hardware. The software may include the compiler 110 and the hardware may include the JSM 102. In accordance with preferred embodiments of the invention, the JSM 102 may execute at least two instruction sets. One instruction set may comprise standard Java bytecodes. As is well-known, Java bytecode is a stack-based intermediate language in which instructions generally target a stack. For example, an integer add ("IADD") Java instruction pops two integers off the top of the stack, adds them together, and pushes the sum back on the stack. As will be explained in more detail below, the JSM 102 comprises a stack-based architecture with various features that accelerate the execution of stack-based Java code, where the stack may include multiple portions that exist in different physical locations.

[0018] Another instruction set executed by the JSM 102 may include instructions other than standard Java instructions. In accordance with at least some embodiments of the invention, other instruction sets may include register-based and memory-based operations to be performed. This other instruction set generally complements the Java instruction set and, accordingly, may be referred to as a complementary instruction set architecture ("C-ISA"). By complementary, it is meant that the execution of more complex Java bytecodes may be substituted by a "micro-sequence" comprising one or more C-ISA instructions that permit address calculation to readily "walk through" the JVM data structures. A micro-sequence also may include one or more

bytecode instructions. The execution of Java may be made more efficient and run faster by replacing some sequences of bytecodes by preferably shorter and more efficient sequences of C-ISA instructions. The two sets of instructions may be used in a complementary fashion to obtain satisfactory code density and efficiency. As such, the JSM 102 generally comprises a stack-based architecture for efficient and accelerated execution of Java bytecodes combined with a register-based architecture for executing register and memory based C-ISA instructions. Both architectures preferably are tightly combined and integrated through the C-ISA.

[0019] Figure 2 shows an exemplary block diagram of the JSM 102. As shown, the JSM includes a core 120 coupled to a data storage 122 and an instruction storage 130. Storage 122 and 130 are preferably integrated, along with core 120, on the same JSM chip. Integrating storage 122 and 130 on the same chip as the core 120 may reduce data transfer time from storage 122 and 130 to the core 120. The core 120 may include one or more components as shown. Such components preferably include a plurality of registers 140, three address generation units ("AGUs") 142, 147, micro-translation lookaside buffers (micro-TLBs) 144, 156, a multi-entry micro-stack 146, an arithmetic logic unit ("ALU") 148, a multiplier 150, decode logic 152, and instruction fetch logic 154. In general, operands may be retrieved from a main stack and processed by the ALU 148, where the main stack may include multiple portions that exist in different physical locations. For example, the main stack may reside in external memory 106 and/or data storage 122. Selected entries from the main stack may exist on the micro-stack 146. In this manner, selected entries on the micro-stack 146 may represent the most current version of the operands in the system 100. Accordingly, operands in external memory 106 and data storage 122 may not be coherent with the versions contained on the micro-stack 146. A plurality of flags 158 preferably are coupled to the micro-stack 146, where the flags 158 indicate the validity of data on the micro-stack 146 and

whether data on the micro-stack 146 has been modified. Also, stack coherency operations may be performed by examining the flags 158 and updating the main stack with valid operands from the micro-stack 146.

[0020] The micro-stack 146 preferably comprises, at most, the top *n* entries of the main stack that is implemented in data storage 122 and/or external memory 106. The micro-stack 146 preferably comprises a plurality of gates in the core 120 of the JSM 102. By implementing the micro-stack 146 in gates (e.g., registers) in the core 120 of the JSM 102, access to the data contained on the micro-stack 146 is generally very fast. Therefore data access time may be reduced by providing data from the micro-stack 146 instead of the main stack. General stack requests are provided by the micro-stack 146 unless the micro-stack 146 cannot fulfill the stack requests. For example, when the micro-stack 146 is in an overflow condition or when the micro-stack 146 is in an underflow condition (as will be described below), general stack requests may be fulfilled by the main stack. By analyzing trends of the main stack, the value of *n,* which represents the size of the micro-stack 146, may be optimized such that a majority of general stack requests are fulfilled by the micro-stack 146, and therefore may provide requested data in fewer cycles. As a result, power consumption of the system 102 may be reduced. Although the value of *n* may vary in different embodiments, in accordance with at least some embodiments, the value of *n* may be the top eight entries in the main stack. In this manner, about 98% of the general stack accesses may be provided by the micro-stack 146, and the number of accesses to the main stack may be reduced.

[0021] Instructions may be fetched from instruction storage 130 by fetch logic 154 and decoded by decode logic 152. The address generation unit 142 may be used to calculate addresses based, at least in part on data contained in the registers 140. The AGUs 142 may calculate addresses for C-ISA instructions. The AGUs 142 may support parallel data accesses for C-ISA instructions that

perform array or other types of processing. AGU 147 couples to the micro-stack 146 and may manage overflow and underflow conditions on the micro-stack 146 preferably in parallel. The micro-TLBs 144, 156 generally perform the function of a cache for the address translation and memory protection information bits that are preferably under the control of the operating system running on the MPU 104.

[0022] Referring now to Figure 3, the registers 140 may include 16 registers designated as R0-R15. Registers R0-R3, R5, R8-R11 and R13-R14 may be used as general purposes ("GP") registers usable for any purpose by the programmer. Other registers, and some of the GP registers, may be used for specific functions. For example, registers R4 and R12 may be used to store two program counters. Register R4 preferably is used to store the program counter ("PC") and register R12 preferably is used to store a micro-program counter ("micro-PC"). In addition to use as a GP register, register R5 may be used to store the base address of a portion of memory in which Java local variables may be stored when used by the current Java method. The top of the micro-stack 146 is reflected in registers R6 and R7. The top of the micro-stack 146 has a matching address in external memory 106 pointed to by register R6. The operands contained on the micro-stack 146 are the latest updated values, while their corresponding values in external memory 106 may or may not be up to date. Register R7 provides the data value stored at the top of the micro-stack 146. Registers R8 and R9 may also be used to hold an address index 0 ("AI0") and an address index 1 ("AI1"), which may be used in calculating addresses in memory generated by various bytecodes, for example, the result of an IADD instruction. Register R14 may also be used to hold the indirect register index ("IRI") that also may be used in calculating memory addresses. Register R15 may be used for status and control of the JSM 102. As an example, one status/control bit (called the "Micro-Sequence-Active" bit) may indicate if the JSM 102 is executing a "simple" instruction or a

"complex" instruction through a "micro-sequence." This bit controls in particular, which program counter is used R4 (PC) or R12 (micro-PC) to fetch the next instruction. A "simple" bytecode instruction is generally one in which the JSM 102 may perform an immediate operation either in a single cycle (e.g., an IADD instruction) or in several cycles (e.g., "dup2_x2"). A "complex" bytecode instruction is one in which several memory accesses may be required to be made within the JVM data structure for various verifications (e.g., NULL pointer, array boundaries). Because these data structure are generally JVM-dependent and thus may change from one JVM implementation to another, the software flexibility of the micro-sequence provides a mechanism for various JVM optimizations now known or later developed.

[0023] The second, register-based, memory-based instruction set may comprise the C-ISA instruction set introduced above. The C-ISA instruction set preferably is complementary to the Java bytecode instruction set in that the C-ISA instructions may be used to accelerate or otherwise enhance the execution of Java bytecodes.

[0024] The ALU 148 adds, subtracts, and shifts data. The multiplier 150 may be used to multiply two values together in one or more cycles. The instruction fetch logic 154 generally fetches instructions from instruction storage 130. The instructions may be decoded by decode logic 152. Because the JSM 102 is adapted to process instructions from at least two instruction sets, the decode logic 152 generally comprises at least two modes of operation, one mode for each instruction set. As such, the decode logic unit 152 may include a Java mode in which Java instructions may be decoded and a C-ISA mode in which C-ISA instructions may be decoded.

[0025] The data storage 122 generally comprises data cache ("D-cache") 124 and data random access memory ("D-RAM") 126. Reference may be made to copending applications U.S. Serial nos. 09/591,537 filed June 9, 2000 (atty docket TI-29884), 09/591,656 filed June 9, 2000 (atty

docket TI-29960), and 09/932,794 filed August 17, 2001 (atty docket TI-31351), all of which are incorporated herein by reference. The main stack, arrays and non-critical data may be stored in the D-cache 124, while Java local variables, critical data and non-Java variables (e.g., C, C++) may be stored in D-RAM 126. The instruction storage 130 may comprise instruction RAM ("I-RAM") 132 and instruction cache ("I-cache") 134. The I-RAM 132 may be used for "complex" micro-sequenced bytecodes or micro-sequences or predetermined sequences of code, as will be described below. The I-cache 134 may be used to store other types of Java bytecode and mixed Java/C-ISA instructions.

[0026] As noted above, the C-ISA instructions generally complement the standard Java bytecodes. For example, the compiler 110 may scan a series of Java bytes codes 112 and replace one or more of such bytecodes with an optimized code segment mixing C-ISA and bytecodes and which is capable of more efficiently performing the function(s) performed by the initial group of Java bytecodes. In at least this way, Java execution may be accelerated by the JSM 102.

[0027] As noted above, the micro-stack 146 includes a finite number of entries, and therefore overflow and underflow conditions may occur. Figures 4A-C depict an overflow condition of the micro-stack 146. Note that although the micro-stack 146 shown in Figures 4A-C is shown containing four entries, preferred embodiments may have any number of entries. As shown in Figure 4A, the micro-stack 146 may include data values or operands A and B, for example as the result of pushing A and B on the micro-stack 146. Stack pointers 162 and 164 reflect the top of the micro-stack 146 and the top of the main stack 160 respectively. When new data values are pushed on the micro-stack 146, the flags 158 may be enabled (indicated by EN in the Figures) to indicate that the new data is valid. A lack of EN in a flag 158 indicates invalid data. Operands pushed on the micro-stack 146 generally are not pushed on a main stack 160. Operands A' and B' indicate the

place of the data in the main stack, but these entries are not coherent with the corresponding micro-stack entries A and B. Coherence may be achieved if A and B are written to main memory during an overflow condition or flushing, as explained below. The main stack pointer 164 is updated at every push or pop. As indicated above, the main stack 160 may exist in external memory 106 and/or data storage 122, and the main stack 160 may be larger than the micro-stack 146.

[0028] Figure 4B shows operands C and D pushed on the micro-stack 146, where the micro-stack 146 is now full. Operands are pushed on the micro-stack 146 in a cyclical manner such that when the micro-stack 146 is full, the data at the bottom of the micro-stack 146 (which in this example is operand A) is overwritten. As operands are pushed into entries of the micro-stack 146, the flag 158 associated with each entry may be checked for validity. If the flag 158 indicates that the data in an entry, where a new push is performed, is valid (i.e., flag 158 enabled), then the entry must be copied on the main stack 160 prior to pushing the next data operand on the micro-stack 146. Figure 4C depicts the result of pushing operand E on the full micro-stack 146 shown in Figure 4B. When the micro-stack 146 is full (shown in Figure 4B), the bottom of the micro-stack 146 is moved into the main stack 160 at an address value equal to the stack pointer 164 minus the number of entries *n* on the micro-stack 146. For example, Figure 4B shows the micro-stack 146 including 4 entries and the stack pointer 164 indicating the top of the main stack 160. In this example, prior to overwriting operand A on the micro-stack 146 with operand E, operand A is copied to an address that is four entries less than the address indicated by the stack pointer 164. Thus, operand A from the micro-stack 164 is written to the main stack as indicated in Figure 4C.

[0029] Figures 5A-C depict an underflow condition of the micro-stack 146 shown in Figure 4C. Referring to Figure 5A, operand E is popped off the micro-stack 146 and then operand D is popped off the micro-stack 146. As operands are popped off of the micro-stack 146 the corresponding flag

158 is invalidated and the stack pointers 162 and 164 are decremented. Figure 5B illustrates a bytecode that provides operands B and C to the ALU 148. The ALU 148 produces a result Z, which is placed back on the micro-stack 146, and the flag 158 is enabled as shown. If a subsequent bytecode requires operand Z as well as another operand that is not on the micro-stack 146, an underflow occurs. Flag 158 is preferably checked for valid data prior to executing bytecodes to determine whether the required data is present on the micro-stack 146. For example, Figure 5C depicts a bytecode requiring operand Z in addition to operand A, which is not on the micro-stack 146. Since flag 158 associated with operand A is not enabled in Figure 5B, operand A is fetched from the main stack 160. In some embodiments, multiple operands may be fetched simultaneously from the main stack 160. In addition, other embodiments include pre-fetching the operands from the main stack 160.

[0030] Flags 158 may include a register with bits allocated for each entry in the micro-stack 146, or alternatively flags 158 may include a read pointer and a write pointer. The read pointer is preferably updated on each stack instruction execution. For example, during an IADD instruction, the read pointer may decrement itself once for each operand that is popped off the stack, and then increment itself once to write the result of the operand back on the stack. The write pointer is preferably updated during an underflow or an overflow. By comparing the values of the read pointer and the write pointer, overflow and underflow conditions can be detected.

[0031] As noted previously, system 100 may be implemented as a mobile cell phone such as that shown in Figure 6. As shown, a mobile communication device includes an integrated keypad 412 and display 414. The JSM 102 and MPU 104 and other components may be included in electronics package 410 connected to the keypad 412, display 414, and radio frequency ("RF") circuitry 416. The RF circuitry 416 may be connected to an antenna 418.

[0032] While the preferred embodiments of the present invention have been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit and teachings of the invention. The embodiments described herein are exemplary only, and are not intended to be limiting. Many variations and modifications of the invention disclosed herein are possible and are within the scope of the invention. Accordingly, the scope of protection is not limited by the description set out above. Each and every claim is incorporated into the specification as an embodiment of the present invention.